DB-Nets: On the Marriage of Colored Petri Nets and Relational Databases

Marco Montali(✉) and Andrey Rivkin

Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy
{montali,rivkin}@inf.unibz.it

Abstract. The integrated management of business processes and master data is being increasingly considered as a fundamental problem, by both the academia and the industry. In this position paper, we focus on the foundations of the problem, arguing that contemporary approaches struggle to find a suitable equilibrium between data- and process-related aspects. We then propose a new formal model, called db-nets, that balances such two pillars through the marriage of colored Petri nets and relational databases. We invite the research community to build on this new model, discussing in particular its potential in conceptual modeling, formal verification, and simulation.

1 Introduction

In contemporary organizations, the integrated management of business processes (BPs) and master data is being increasingly considered as a fundamental problem, both by academia and the industry. From the practical point of view, it has been widely recognized that the traditional isolation between process and data management induces fragmentation and redundancies in the organizational structure and its underlying IT solutions, with experts and tools solely centered around data, and others only focusing on process management [19,30,31]. This isolation falls short, especially when it comes to knowledge-intensive and human-empowered processes [4,21,23].

State-of-the-art BP management systems (BPMSs), such as Bizagi BPM, Bonita BPM, Activiti, Camunda, and YAWL[1], actually provide clean conceptualizations for the process control flow as well as the "touching joints" between control flow and data: *(i)* process instances (also called cases) carry local data, *(ii)* a database backend is typically used to store global, persistent data, *(iii)* the decision logic queries local and persistent data to choose which path to select among multiple alternatives, *(iv)* the task logic dictates how to update local and persistent data. However, as argued in [17], no well-established approach exists to express the decision and task logic, which is in fact handled in an ad-hoc way, usually combining tool-specific languages with general purpose programming languages such as Java. The result is that the interaction of the process and its data becomes a sort of "procedural attachment" that is exploited during

[1] bizagi.com, bonitasoft.com, activiti.org, camunda.com, yawlfoundation.org.

M. Koutny et al. (Eds.): ToPNoC XII, LNCS 10470, pp. 91–118, 2017.
DOI: 10.1007/978-3-662-55862-1_5

the process enactment, but that is not conceptually well-understood [11]. As an effect, the verification tasks offered by such systems become either disabled when data are present, or produce misleading answers, since they do not take into account that the presence of data subtly affects the behaviors described by the process [17]. For example, seemingly concurrent behavior in the process may be in fact sequenced due to the presence of data constraints, which implicitly induce an order on the allowed data updates. More generally, non-executable paths and deadlocks may emerge only when the interplay between the process and its data is considered.

Foundational research has also witnessed a similar separation, with non-interacting areas of research either focused on data management or dynamic concurrent systems, with database theory and Petri net theory being the two most prominent representatives of each field. Over the years, both fields actually entered into the problem of combining data and processes, with quite complementary approaches. A first series of approaches stem from Petri nets, the reference formalism to represent the control-flow of BPs. All such models are more or less directly inspired by Colored Petri nets (CPNs) [3,22], where colors abstractly account for data types, and where the control threads (i.e., tokens) traversing the net carry data conforming to colors. Verification in this setting is tackled by severely restricting the contribution of data. This is done by requiring colors to have a finite domain, thus realizing a form of a-priori propositionalization of the data, or by limiting the way tokens can carry data. This latter approach has led to the identification of several CPN fragments that are amenable to formal analysis even in the case of infinite color domains, ranging from nets where tokens carry single data values (as in data- and ν-nets [25,32]), to nets where tokens are associated to more complex data structures such as nested relations [20], nested terms [34], or XML documents [8]. However, the common limitation of all such approaches is that data are still subsidiary to the control-flow dimension: data elements are "locally" attached to tokens, while no native support for global, persistent relational data is provided. In this light, CPNs naturally support cases and case data through the abstraction of colored tokens [33]. However, they do not lend themselves to modeling, querying, updating, and ultimately reasoning on persistent, relational data, like those typically maintained inside an enterprise information system. For this reason, they are unable to impact contemporary BPMSs, which, as argued above, all support the explicit linkage of BPs and an underlying persistent relational layer [17].

The second group of foundational approaches to data-aware processes has emerged at the intersection of database theory, formal methods and conceptual modeling, and specularly mirrors the advantages and lacks of CPN-based solutions. Such proposals go under the umbrella term of data-centric approaches [11], and gained momentum during the last decade, in particular due to the development of the business artifact paradigm [15], which also lead to concrete languages and implementations [16,23]. The common denominator of all such approaches is that processes are centered around an explicit, persistent data component maintaining information about the domain of interest, and possibly capturing

also its semantics in terms of classes, relations, and constraints. Atomic tasks induce CRUD (create-ready-update-delete) operations over the data component, in turn supporting the evolution of the master data maintained therein. Proposals then differ in terms of the adopted data model (e.g., relational, tree-shaped, graph-structured), and on the nature of information (e.g., whether it is complete or not). For example, [9,18] focus on relational databases, while [7] on XML and tree-shaped data models. The main downside of data-centric process models is that they disregard an explicit representation of how tasks have to be sequenced over time, only implicitly representing the control flow via (event-)condition-action rules [9,16,18]. Hence, they are too distant from contemporary BPMSs, which all rely on Petri net-inspired languages to define the process control flow.

We believe that this lack of equilibrium is a major obstacle towards the adoption of such foundational results into contemporary BPMSs, and that a more balanced formal model will pave the way towards simulation, verification, monitoring, and mining techniques that more effectively reflect, and exploit, the main abstractions offered by contemporary BPMSs, and their interrelationships. Technically, this in turn calls for the development of a formal model that natively establishes intimate, synergic connections between CPNs and data-centric approaches. To the best of our knowledge, the only existing proposal that makes an effort in this direction is [17]. However, it employs workflow nets [1] for capturing the process control flow, without leveraging the sophistication of CPNs. Taking inspiration from [17], we then propose *db-nets* , a new, balanced formal model for data-aware processes, rooted in CPNs and relational databases. We rigorously describe the abstractions offered by the model, and formalize its execution semantics. We finally invite the research community to build on this new model, discussing its potential impact on modeling, verification, and simulation. In particular, although preliminary, the verification results here presented introduce conditions that could not be singled out in previous formal models.

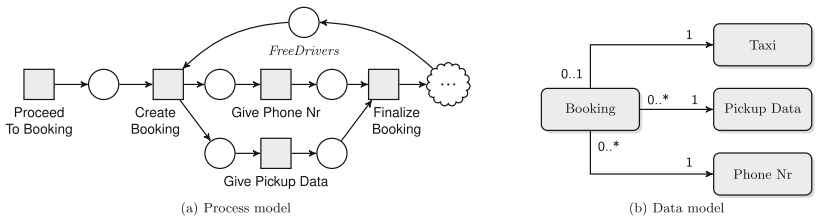


Fig. 1. A Petri net (a) and an informal diagram (b) respectively capturing the process and the data for our taxi booking example

2 The DB-Net Model: A Gentle Introduction

We discuss a concise, yet meaningful example that serves a twofold purpose: illustrating the issues arising when separating data and process modeling, and providing a gentle introduction to our proposal.

An airport website offers a door-to-door taxi shuttle service that can be booked on-line without any registration. To book a taxi, one only needs to leave a phone number, a pickup address and a desired pickup time. Once all the necessary data have been provided, the client confirms the booking and a free taxi driver is assigned to execute the order. In a typical industrial setting where business process experts and master data managers operate within separate silos [12,31], capturing this scenario would require to independently gather process and data requirements about the booking process. The *process expert* uses Petri nets (see Fig. 1(a)) to capture the process requirements as the basis for the construction of a web application. The application consists of the following steps. Whenever a client enters a taxi booking page, a booking is created, by non-deterministically picking a free taxi driver that will serve the booking (this is done by consuming a token from the resource place *FreeDrivers*). The process then demands the user to provide the relevant booking data (phone number and pickup data). After that, the booking is finalized, entering into the phase of the process where the booking is actually served, eventually leading to free the taxi driver (this is modeled by feeding a token back to the *FreeDrivers* place).

On the other hand, the *master data expert* typically gathers requirement about relevant data in the domain, and how to structure them in terms of classes, relationships, and constraints that must hold in each system snapshot. This creates the basis for building a corresponding database schema. Figure 1(b) sketches the resulting diagram informally, showing that a booking requires a combination of taxi, pickup data, and phone number, while, on the other hand, at any moment a taxi may be associated to one or no booking (i.e., busy or free).

Both models are reasonable in their respective contexts. However, their combination may lead to an overall faulty solution. For example, once the process in Fig. 1(a) is deployed on top of the database obtained from the diagram in Fig. 1(b), the process expert may be tempted to program the task logic underlying Create Booking by actually creating a new instance of class *Booking*. However, this update would be rejected by the underlying database, since its schema stipulates that a booking can exist only if all the information related to taxi, pickup data, and phone number is provided all at once.

Fixing this mismatch is possible only by simultaneously understanding the process and the data schema. One possible solution is to let the process create a booking during the execution of Create Booking, relaxing to "0...1" all multiplicity constraints of type "1" in Fig. 1(b). This would in fact allow the data model to store an incomplete booking where only some booking-related information is known. Another possible solution would instead require to change the task logic of the process, for example introducing local data variables to keep track of the reserved taxi, provided phone number, and provided pickup data, then creating a booking entry in the underlying database during the execution of the Finalize Booking task.

To enable this form of integrated modeling and analysis, we propose db-nets. The main idea is to maintain both the data and the process model intact, and to enrich them with an interface that conceptually interconnects them. This three-layered approach, applied to the booking example, may lead to the solution

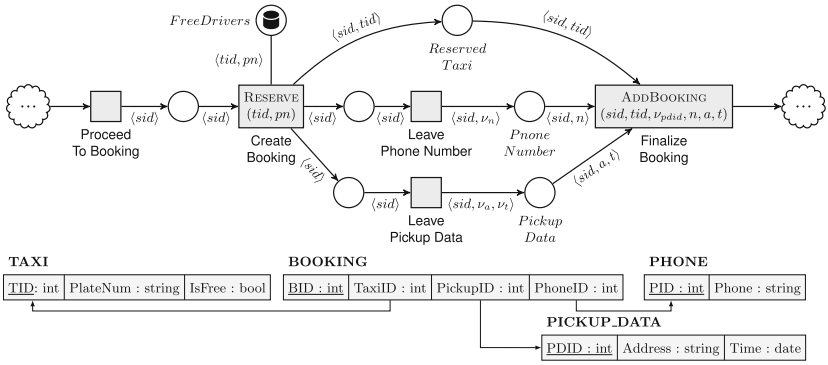


Fig. 2. A db-net representing the taxi booking process

shown in Fig. 2. The first layer, shown at the bottom, is the *persistence layer*, a full-fledged relational database with constraints (in the figure, primary and foreign keys) that faithfully mirrors the data model of Fig. 1(b). The second layer, shown at the top, is the *control layer*, which captures the process logic using a sophisticated variant of a CPN, which supports: *(i)* typing of tokens, so as to account for local variables attached to execution threads; *(ii)* injection of possibly fresh data values via special so-called ν-variables (leveraging the ν-PN model [32]); *(iii)* accessing the content of the underlying data layer via special *view-places*; *(iv)* updating the underlying data layer by attaching a database update logic to its transitions.

Intuitively, *sid* – a variable used to manage booking sessions (resembling the classical notion of "case id" in BPM) – will have a type **int**, while the *PickupData* place will host tokens carrying data of type $\mathbf{int} \times \mathbf{string} \times \mathbf{date}$. The only view-place *FreeDrivers* accesses the underlying data layer to know which free taxies do exist when the Create Booking transition is fired. The connection between the view place and the transition is a read arc: to realize a clean update logic for the data, tuples obtained from the data layer are not consumed at the level of the net, but are manipulated via the update logic attached to the net transitions.

Such query and update functionalities are offered by a third, intermediate layer in our framework, called *data logic layer*. On the one hand, view places exploit the data logic layer to query the underlying data layer. E.g., the *Free Drivers* place exploits a query that returns the IDs of taxis whose *isFree* column is true. This realizes the fact that Create Booking cannot fire if no ID of this kind exists. On the other hand, the two transitions Create Booking and Finalize Booking exploit the data logic layer to update the persistent data depending on the current state of the net, the data locally carried by tokens, and additional data obtained from the external world via additional variables. Using this information, the transitions call corresponding parametric actions that are exposed by the data logic layer, and that encapsulate the update logic. In our example, whenever Create Booking fires, action Reserve will update selected taxi setting its *isFree*

column to false, thus realizing a form of "pre-booking" for the taxi. From now on, the corresponding taxi ID will not be returned anymore by the query feeding the *Free Drivers* view place. Finally, when Finalize Booking fires, all booking-related data, so far only locally attached to tokens, are fed to action AddBooking, creating a new persistent booking into the persistence layer.

Notice that this solution preserves the intention of the original models from Fig. 1: the data are kept within the net until the point when they can be combined into a proper tuple to be inserted into the underlying database, without violating its foreign key constraints. In addition, the db-net clearly separates the control flow of the process from persistent resources. While in Fig. 1(a) the taxi resources have to be explicitly defined in advance and enumerated in the net, in our db-net they are delegated to the databases, and whenever the actual fleet of taxis changes, this change will be simply recorded in the data layer.

3 The db-net Formal Model

In this section, we formalize db-nets going through the three layers informally introduced in Sect. 2: persistence layer, data logic layer, and control layer.

3.1 Persistence Layer

The persistence layer maintains the relevant data in the domain of interest. To this end, we rely on standard relational databases equipped with constraints, in the spirit of [9]. First-order (FO) constraints allow for the formalization of conventional database constraints, such as keys and functional dependencies, as well as semantic constraints reflecting the domain of interest. Differently from [9], though, we also consider data types, on the one hand resembling concrete logical schemas of relational databases (where table columns are typed), and on the other reconciling the persistence layer with the notion of "color" in CPNs.

Definition 1. A *data type* $\mathcal{D}$ is a pair $\langle \Delta_{\mathcal{D}}, \Gamma_{\mathcal{D}} \rangle$, where $\Delta_{\mathcal{D}}$ is a *value domain*, and $\Gamma_{\mathcal{D}}$ is a finite set of *predicate symbols*. Each predicate symbol $S \in \Gamma_{\mathcal{D}}$ comes with an arity n_S and an n-ary predicate $S^{\mathcal{D}} \subseteq \Delta_{\mathcal{D}}^n$ that rigidly defines its semantics. A *type domain* is a finite set of data types. □

In the following, we use $\mathfrak{D}$ to denote a type domain of interest, assuming that types in $\mathfrak{D}$ are pairwise disjoint, that is, their domains do not intersect, and their predicate symbols are syntactically distinguished. This guarantees that a predicate symbol S defined in some type of $\mathfrak{D}$, is defined only in that type, which can be then unambiguously denoted, with slight abuse of notation, by $\texttt{type}(S)$. We also employ $\Delta_{\mathfrak{D}} = \bigcup_{\mathcal{D} \in \mathfrak{D}} \Delta_{\mathcal{D}}$. Examples of data types are:

- **string** : $\langle \mathbb{S}, \{=_s\} \rangle$, strings with the equality predicate;
- **real** : $\langle \mathbb{R}, \{=_r, <_r\} \rangle$, real numbers with the usual comparison operators;
- **int** : $\langle \mathbb{Z}, \{=_{int}, <_{int}, succ\} \rangle$, integers with the usual comparison operators, as well as the successor predicate.

Definition 2. A $\mathfrak{D}$-*typed relation schema* is a pair $\langle R, \vec{\mathcal{D}} \rangle$, where R is a relation name, and $\vec{\mathcal{D}}$ is a tuple of elements from $\mathfrak{D}$, indicating the data types associated to each component of R. A $\mathfrak{D}$-*typed database schema* $\mathcal{R}$ is a finite set of $\mathfrak{D}$-typed relation schemas. □

For compactness, we represent a typed relation schema $\langle R, \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle \rangle$ using notation $R(\mathcal{D}_1, \ldots, \mathcal{D}_n)$.

Definition 3. Given a $\mathfrak{D}$-typed database schema $\mathcal{R}$, a $\mathfrak{D}$-*typed database instance* $\mathcal{I}$ *over* $\mathcal{R}$ is a *finite* set of facts of the form $R(\mathtt{o}_1, \ldots, \mathtt{o}_n)$, such that *(i)* $R(\mathcal{D}_1, \ldots, \mathcal{D}_n) \in \mathcal{R}$ and *(ii)* $\mathtt{o}_i \in \Delta_{\mathcal{D}_i}$, for each $i \in \{1, \ldots, n\}$. Given a type $\mathcal{D} \in \mathfrak{D}$, the $\mathcal{D}$-*active domain of* $\mathcal{I}$, written $Adom_{\mathcal{D}}(\mathcal{I})$, is the set of values in $\Delta_{\mathcal{D}}$ such that $\mathtt{o} \in Adom_{\mathcal{D}}(\mathcal{I})$ if and only if $\mathtt{o} \in \Delta_{\mathcal{D}}$ and $\mathtt{o}$ occurs in $\mathcal{I}$. We also define $Adom_{\mathfrak{D}}(\mathcal{I}) = \bigcup_{\mathcal{D} \in \mathfrak{D}} Adom_{\mathcal{D}}(\mathcal{I})$. □

Example 1. The relation schema for a taxi in Fig. 1 is $Taxi(\mathbf{int}, \mathbf{string}, \mathbf{bool})$. Then, $Taxi(\mathtt{1}, \mathtt{123AB}, \mathsf{false})$ is a fact for *Taxi* denoting that taxi number `1` has plate number `123AB` and is currently busy. ■

We now turn to queries. As query language, we resort to standard first-order logic (FOL), interpreted under the active domain semantics [26]. This means that quantifiers are relativized to the active domain of the database instance of interest, guaranteeing that queries are domain-independent (actually, safe-range): their evaluation only depends on the values explicitly appearing in the database instance over which they are applied. Recall that this query language is equivalent to the well-known SQL standard [6]. Since the relational structures we consider are typed, the logic is typed as well.

Given a type domain $\mathfrak{D}$, we fix a countably infinite set $\mathcal{V}_{\mathfrak{D}}$ of variables. Each variable is typed. To this end, we introduce a *variable typing function* $\mathtt{type} : \mathcal{V}_{\mathfrak{D}} \to \mathfrak{D}$ mapping variables to their types. The typing function prescribes that x may be substituted only by values taken from $\Delta_{\mathtt{type}(x)}$. For compactness, the variable type may be explicitly shown using a colon notation $x{:}\mathtt{type}(x)$.

Definition 4. A *(well-typed)* $\mathtt{FO}(\mathfrak{D})$ *query* over a $\mathfrak{D}$-typed database schema $\mathcal{R}$ is a formula of the form:

$$Q \ ::= \ S(\vec{y}) \mid R(\vec{z}) \mid \neg Q \ \mid \ Q_1 \wedge Q_2 \ \mid \ \exists x.Q, \text{ where}$$

- for $\vec{y} = \langle y_1, \ldots, y_n \rangle$, we have that S/n is a predicate defined in $\Gamma_{\mathcal{D}}$ for some $\mathcal{D} \in \mathfrak{D}$, and for each $i \in \{1, \ldots, n\}$, we have that y_i is either a value $\mathtt{o} \in \Delta_{\mathcal{D}}$, or a variable $x \in \mathcal{V}_{\mathfrak{D}}$ with $\mathtt{type}(x) = \mathcal{D}$;
- for $\vec{z} = \langle z_1, \ldots, z_m \rangle$, we have that $R(\mathcal{D}_1, \ldots, \mathcal{D}_m)$ is a relation defined in $\mathcal{R}$, and for each $i \in \{1, \ldots, m\}$, we have that z_i is either a value $\mathtt{o} \in \Delta_{\mathcal{D}_i}$, or a variable $x \in \mathcal{V}_{\mathfrak{D}}$ with $\mathtt{type}(x) = \mathcal{D}_i$.

We use standard abbreviations $Q_1 \vee Q_2 = \neg(\neg Q_1 \wedge \neg Q_2)$, and $\forall x.Q = \neg\exists x.\neg Q$. □

Definition 5. A variable $x \in \mathcal{V}_{\mathfrak{D}}$ is *free* in a $\texttt{FO}(\mathfrak{D})$ query Q, if x occurs in Q but is not in the scope of any quantifier. By $\mathit{Free}(Q)$ we denote the set of variables occurring free in Q. A *boolean query* is a query without free variables. □

Given a query Q such that $\mathit{Free}(Q) = \{x_1, \ldots, x_n\}$, we employ notation $\texttt{Q}_{\texttt{name}}(x_1, \ldots, x_n) \coloneq Q$ to emphasize the free variables of Q, and to fix a natural ordering over them.

Example 2. Consider the *Taxi* relation schema of Example 1. Query $\texttt{Q}_{\texttt{FreeTaxiId}}(x) \coloneq \exists p.\, \mathit{Taxi}(x, p, \texttt{true})$ returns the set of ids associated to free taxies. ■

As usual, queries are used to extract answers from a database instance.

Definition 6. Given a set $X = \{x_1, \ldots, x_n\}$ of typed variables, a *substitution* for X is a function $\theta : X \to \Delta_{\mathfrak{D}}$ mapping variables from X into values, such that for every $x \in X$, we have $\theta(x) \in \Delta_{\texttt{type}(x)}$. A *substitution θ for a $\texttt{FO}(\mathfrak{D})$ query Q* is a substitution for the free variables of Q. □

As customary, we may view a substitution θ for a query Q simply as a tuple of values, assuming the natural ordering over the free variables of Q. We denote by $Q\theta$ the boolean query obtained from Q by replacing each free variable $x \in \mathit{Free}(Q)$ with the corresponding value $\theta(x)$. In the following, we apply substitutions to any structure containing variables. Substitutions are the basis for capturing the semantics of query answers, which we tackle next.

Definition 7. Given a $\mathfrak{D}$-typed database schema $\mathcal{R}$, a $\mathfrak{D}$-typed instance $\mathcal{I}$ over $\mathcal{R}$, A $\texttt{FO}(\mathfrak{D})$ query Q over $\mathcal{R}$, and a substitution θ for Q, we inductively define relation $\mathcal{I}$ *entails* Q *under* θ *with active domain semantics*, written $\mathcal{I}, \theta \models Q$, as:

$$
\begin{array}{lll}
\mathcal{I}, \theta \models R(y_1, \ldots, y_n) & \text{if} & R(y_1, \ldots, y_n)\theta \in \mathcal{I} \\
\mathcal{I}, \theta \models S(y_1, \ldots, y_n) & \text{if} & S(y_1, \ldots, y_n)\theta \in S^{\texttt{type}(S)} \\
\mathcal{I}, \theta \models \neg Q & \text{if} & \mathcal{I}, \theta \not\models Q \\
\mathcal{I}, \theta \models Q_1 \wedge Q_2 & \text{if} & \mathcal{I}, \theta \models Q_1 \text{ and } \mathcal{I}, \theta \models Q_2 \\
\mathcal{I}, \theta \models \exists x.Q & \text{if} & \text{there exists } \texttt{o} \in \mathit{Adom}_{\texttt{type}(x)}(\mathcal{I}) \text{ such that } \mathcal{I}, \theta[x/\texttt{o}] \models Q
\end{array}
$$

where $\theta[x/\texttt{o}]$ denotes the substitution obtained from θ by assigning $\texttt{o}$ to x.[2] □

Definition 8. Given a $\mathfrak{D}$-typed database schema $\mathcal{R}$, a $\mathfrak{D}$-typed instance $\mathcal{I}$ over $\mathcal{R}$, and a $\texttt{FO}(\mathfrak{D})$ query $Q(x_1, \ldots, x_n)$ over $\mathcal{R}$, the set of *answers to Q in $\mathcal{I}$*, written $\mathit{ans}(Q, \mathcal{I})$, is the set of substitutions θ from the free variables of Q to the active domain of $\mathcal{I}$, such that Q holds in $\mathcal{I}$ under θ:

$$\mathit{ans}(Q, \mathcal{I}) = \{\text{substitution } \theta : \mathit{Free}(Q) \to \mathit{Adom}_{\mathfrak{D}}(\mathcal{I}) \mid \mathcal{I}, \theta \models Q\}$$

□

[2] If $\theta(x)$ is defined, its value is replaced by $\texttt{o}$, otherwise θ is extended so that $\theta(x) = \texttt{o}$.

When Q is boolean, we write $ans(Q, \mathcal{I}) \equiv \mathsf{true}$ if $\langle\rangle \in ans(Q, \mathcal{I})$, or $ans(Q, \mathcal{I}) \equiv \mathsf{false}$ if $ans(Q, \mathcal{I}) = \emptyset$.

Example 3.
Let $\mathcal{I}_t = \{Taxi(1, 123AB, \mathsf{false}), Taxi(2, 432CD, \mathsf{true}), Taxi(3, 456DA, \mathsf{true})\}$ be a database instance for taxies, and consider query $\mathtt{Q}_{\mathtt{FreeTaxi}}(x, y)$:- $Taxi(x, y, \mathsf{true})$, which extracts the id and plate number of all free taxies. We then have $ans(\mathtt{Q}_{\mathtt{FreeTaxi}}(x, y), \mathcal{I}_t) = \{\{x \mapsto 2, y \mapsto 432CD\}, \{x \mapsto 3, y \mapsto 456DA\}\}$. ■

We are finally ready to define the persistence layer.

Definition 9. A $\mathfrak{D}$-typed *persistence layer* is a pair $\langle \mathcal{R}, \mathcal{E} \rangle$ where: *(i)* $\mathcal{R}$ is a $\mathfrak{D}$-typed database schema; *(ii)* $\mathcal{E}$ is a finite set $\{\Phi_1, ..., \Phi_k\}$ of boolean $\mathtt{FO}(\mathfrak{D})$ queries over $\mathcal{R}$, modeling *constraints over* $\mathcal{R}$. □

Example 4. Boolean query $\forall x, y_1, y_2, z_1, z_2.Taxi(x, y_1, z_1) \wedge Taxi(x, y_2, z_2) \rightarrow y_1 = y_2 \wedge z_1 = z_2$ expresses that the first component of *Taxi* (i.e., the taxi id) is a key for the *Taxi* relation schema. ■

The presence of constraints calls for a definition of which database instances are compliant by a given persistence layer, i.e., satisfy its constraints.

Definition 10. Given a $\mathfrak{D}$-typed persistence layer $\mathcal{P} = \langle \mathcal{R}, \mathcal{E} \rangle$ and a $\mathfrak{D}$-typed database instance $\mathcal{I}$, we say that $\mathcal{I}$ *complies with* $\mathcal{P}$ if: *(i)* $\mathcal{I}$ is defined over $\mathcal{R}$; *(ii)* $\mathcal{I}$ satisfies all constraints in $\mathcal{E}$, that is, $ans(\bigwedge_{\Phi \in \mathcal{E}} \Phi, \mathcal{I}) \equiv \mathsf{true}$. □

Example 5. The persistence layer $\mathcal{P} = \langle \mathcal{R}, \mathcal{E} \rangle$ is a fragment of an information system used by a company to handle the submission of tickets, and their management by employees. $\mathcal{R}$ employs types **string** and **int** to define the following relation schemas:

- $Emp(\mathbf{string})$ lists employee (names);
- $Ticket(\mathbf{int}, \mathbf{string})$ models ticket (ids) and their description;
- $Resp(\mathbf{string}, \mathbf{int})$ models which employees handle which tickets: $Resp(\mathtt{e}, 1)$ indicates that the employee named `e` is responsible for ticket number `1`.
- $Log(\mathbf{int}, \mathbf{string}, \mathbf{string})$ represents a log table storing information about all the tickets processed so far, also listing their responsible employees and their description.

The persistence layer is also equipped with a set of constraints over $\mathcal{R}$, expressing (primary) keys, foreign keys, functional dependencies, and multiplicity constraints. E.g., the ticket number provides the primary key for *Ticket*, the second component of *Resp* references the primary key of *Ticket*, and *each employee can handle at most one ticket at a time*. It is well-known that such constraints can be formalized in FO [6]. E.g., the latter constraint may be formalized as: $\forall e, t_1, t_2.Resp(e, t_1) \wedge Resp(e, t_2) \rightarrow t_1 = t_2$. ■

3.2 Data Logic Layer

The data logic layer provides a bidirectional "interface" to interact with a database instance complying with a persistence layer of interest. On the one hand, the data logic allows one to *extract* data from the database instance using queries. On the other hand, it allows one to *update* the database instance, adding and deleting possibly multiple facts at once, with a *transactional* semantics: if the new database instance obtained after the update is still compliant with the persistence layer, the update is *committed*, otherwise it is *rolled back*. This approach is in line with how database management systems operate in practice.

To query the database instance, we use $\texttt{FO}(\mathfrak{D})$ queries as in Definition 4. To update the database instance, we instead resort to the literature on data-centric processes [11,35], where *actions* are typically used to apply CRUD (create-read-update-delete) operations over a relational database. Specifically, we adopt a minimalistic approach, keeping the actions as simple as possible. The approach is inspired by the well-known STRIPS language for planning, which has been adopted also in for data-centric processes [5]. More sophisticated forms of actions, as those in [9], can be seamlessly introduced.

Definition 11. A *(parameterized) action* over a $\mathfrak{D}$-typed persistence layer $\langle \mathcal{R}, \mathcal{E} \rangle$ is a tuple $\langle \mathtt{n}, \vec{p}, F^+, F^- \rangle$, where: *(i)* $\mathtt{n}$ is the *action name*; *(ii)* $\vec{p}$ is a tuple of pairwise distinct typed variables from $\mathcal{V}_{\mathfrak{D}}$, denoting the *action (formal) parameters*. *(iii)* F^+ and F^- respectively represent a finite set of $\mathcal{R}$-facts over $\vec{p}$, to be *added* to and *deleted* from the current database instance. Given a typed relation $R(\mathcal{D}_1, \ldots, \mathcal{D}_n) \in \mathcal{R}$, an R-fact over $\vec{p}$ has the form $R(y_1, \ldots, y_n)$, such that for every $i \in \{1, \ldots, n\}$, y_i is either a value $\mathtt{o} \in \Delta_{\mathcal{D}_i}$, or a variable $x \in \vec{p}$ with $\mathtt{type}(x) = \mathcal{D}_i$. An $\mathcal{R}$-fact is an R-fact for some relation R from $\mathcal{R}$. □

To access the different components of an action $\alpha = \langle \mathtt{n}, \vec{p}, F^+, F^- \rangle$, we use a dot notation: $\alpha{\cdot}\mathtt{name} = \mathtt{n}$, $\alpha{\cdot}\mathtt{params} = \vec{p}$, $\alpha{\cdot}\mathtt{add} = F^+$, and $\alpha{\cdot}\mathtt{del} = F^-$.

Example 6. Consider the RESERVE action from Fig. 1. It takes as input two parameters, respectively denoting a taxi id and its plate number, and has the effect of switching its status from free to busy. This is modeled as follows:

$$\text{RESERVE}{\cdot}\mathtt{params} = \langle id, pn \rangle \qquad \begin{array}{l} \text{RESERVE}{\cdot}\mathtt{del} = \{\, \mathit{Taxi}(id, pn, \mathsf{true}) \} \\ \text{RESERVE}{\cdot}\mathtt{add} = \{\, \mathit{Taxi}(id, pn, \mathsf{false}) \} \end{array} \quad \blacksquare$$

We now turn to the semantics of actions. Actions are executed by grounding their parameters to values. Given an action α and a (parameter) substitution θ for α, we call *action instance* $\alpha\theta$ the (ground) action resulting from α by substituting its parameters with corresponding values, as specified by θ.

Definition 12. Let $\mathcal{P} = \langle \mathcal{R}, \mathcal{E} \rangle$ be a $\mathfrak{D}$-typed persistence layer, $\mathcal{I}$ be a $\mathfrak{D}$-typed database instance $\mathcal{I}$ compliant with $\mathfrak{D}$, α be an action over $\mathcal{P}$, and θ be a substitution for *action*$\cdot\mathtt{params}$. The *application* of $\alpha\theta$ on $\mathcal{I}$, written $\mathtt{apply}(\alpha\theta, \mathcal{I})$, is a database instance over $\mathcal{R}$ obtained as $(\mathcal{I} \setminus F^-_{\alpha\theta}) \cup F^+_{\alpha\theta}$, where: *(i)* $F^-_{\alpha\theta} = \bigcup_{R(\vec{y}) \in \alpha\cdot\mathtt{del}} R(\vec{y})\theta$; *(ii)* $F^+_{\alpha\theta} = \bigcup_{R(\vec{y}) \in \alpha\cdot\mathtt{add}} R(\vec{y})\theta$. We say that $\alpha\theta$ can be *successfully applied* to $\mathcal{I}$ if $\mathtt{apply}(\alpha\theta, \mathcal{I})$ complies with $\mathcal{P}$. □

The application of an action instance amounts to ground all the facts contained in the definition of the action as specified by the given substitution, then applying the update on the given database instance, giving priority to additions over deletions (this is a standard approach, which unambiguously handles the situation in which the same fact is asserted to be added and deleted).

Example 7. Consider the data layer shown in Fig. 2, the database instance $\mathcal{I}_t$ from Example 3, and action RESERVE from Example 6. The application of RESERVE$[id/2, pn/\texttt{432CD}]$ is *successful*, and leads to the new database instance $\mathcal{I}'_t = \{Taxi(1, \texttt{123AB}, \texttt{false}), Taxi(2, \texttt{432CD}, \texttt{false}), Taxi(3, \texttt{456DA}, \texttt{true})\}$, where taxi number 2 is in fact busy. ■

The data logic simply exposes a set of queries and a set of actions that can be used by the control layer to obtain data from the persistence layer, and to induce updates on the persistence layer.

Definition 13. Given a $\mathfrak{D}$-typed persistence layer $\mathcal{P}$, a $\mathfrak{D}$-*typed data logic layer over* $\mathcal{P}$ is a pair $\langle \mathcal{Q}, \mathcal{A} \rangle$, where: *(i)* $\mathcal{Q}$ is a finite set of $\texttt{FO}(\mathfrak{D})$ queries over $\mathcal{P}$; *(ii)* $\mathcal{A}$ is a finite set of actions over $\mathcal{P}$. □

Example 8. We make the scenario of Example 5 operational, introducing a data logic layer $\mathcal{L}$ over $\mathcal{P}$. $\mathcal{L}$ exposes two queries to inspect the persistence layer:

- $\texttt{Q}_\texttt{e}(e)$:- $Emp(e) \wedge \neg \exists t.Resp(e, t)$, to extract *idle* employees;
- $\texttt{Q}_\texttt{t}(t, d)$:- $Ticket(t, d)$, to extract tickets and their description.

In addition, $\mathcal{L}$ provides three main functionalities to manipulate tickets in the persistence layer: ticket registration, assignment/release, and logging. Such functionalities are realized through four actions (where, for simplicity, we blur the distinction between an action and its name). The registration of a new ticket is managed by an action REG that, given an integer t, and two strings e and d, (REG·$\texttt{params} = \langle t, e, d \rangle$), simultaneously creates a ticket identified by t and described by d into the persistence layer, and assigns the employee identified by e to such ticket (thus making her *busy*):

$$\text{REG}\cdot\texttt{del} = \{Emp(e)\} \qquad \text{REG}\cdot\texttt{add} = \{Ticket(t, d), Resp(e, t)\}$$

Two specular actions ASSIGN and RELEASE assign or release a ticket to/from an employee, making her busy or idle. Both actions take as input a string for the employee name and an integer for a ticket it (ASSIGN·$\texttt{params}$ = RELEASE·$\texttt{params}$ = $\langle e, t \rangle$), and update e by removing or adding that e is responsible of t:

$$\text{RELEASE}\cdot\texttt{del} = \text{ASSIGN}\cdot\texttt{add} = \{Resp(e, t)\} \qquad \text{RELEASE}\cdot\texttt{add} = \text{ASSIGN}\cdot\texttt{del} = \emptyset$$

Finally, action LOG with LOG·$\texttt{params} = \langle t, e, d \rangle$ is used to flush all the information of a ticket into a log table. The action erases all information about the ticket, and logs that it has been processed, also recalling its employee and description:

$$\text{LOG}\cdot\texttt{del} = \{Ticket(t, d), Resp(e, t)\} \qquad \text{LOG}\cdot\texttt{add} = \{Log(t, e, d)\}$$

■

3.3 Control Layer

The control layer employs a variant of CPNs to capture the process control flow, and how it interacts with an underlying persistence layer through the functionalities provided by the data logic. The spirit is to conceptually ground CPNs by adopting a data-oriented approach. This is done by introducing dedicated constructs exploiting such functionalities, as well as simple, declarative patterns to capture the typical token consumption/creation mechanism of CPNs.

Before introducing the different constitutive elements of the control layer together with their graphical appearance, we fix some preliminary notions. We consider the standard notion of a *multiset*. Given a set A, the *set of multisets* over A, written $A^{\oplus}$, is the set of mappings of the form $m : A \to \mathbb{N}$. Given a multiset $S \in A^{\oplus}$ and an element $a \in A$, $S(a) \in \mathbb{N}$ denotes the number of times a appears in S. Given $a \in A$ and $n \in \mathbb{N}$, we write $a^n \in S$ if $S(a) = n$. We also consider the usual operations on multisets. Given $S_1, S_2 \in A^{\oplus}$: *(i)* $S_1 \subseteq S_2$ (resp., $S_1 \subset S_2$) if $S_1(a) \leq S_2(a)$ (resp., $S_1(a) < S_2(a)$) for each $a \in A$; *(ii)* $S_1 + S_2 = \{a^n \mid a \in A \text{ and } n = S_1(a) + S_2(a)\}$; *(iii)* if $S_1 \subseteq S_2$, $S_2 - S_1 = \{a^n \mid a \in A \text{ and } n = S_2(a) - S_1(a)\}$; *(iv)* given a number $k \in \mathbb{N}$, $k \cdot S_1 = \{a^{kn} \mid a^n \in S_1\}$.[3]

Places. The control layer contains a finite set P of places, which in turn are classified in two groups. On the one hand, so-called *control places* play the role of standard places in classical Petri nets: they represent conditions/states of a dynamic system. On the other hand, so-called *view places* are used as an interface to the underlying persistence layer, so as to make the persistent data available to the control layer. We then have $P = P_c \uplus P_v$, where P_c and P_v respectively denote the set of control and view places.

In the spirit of CPNs, the control layer assigns to each place a color, which in turn combines one or more data types from a type domain $\mathfrak{D}$. Formally, a $\mathfrak{D}$-*color* is a cartesian product $\mathcal{D}_1 \times \ldots \times \mathcal{D}_m$, where for each $i \in \{1, \ldots, m\}$, we have $\mathcal{D}_i \in \mathfrak{D}$. We denote by Σ the set of all possible $\mathfrak{D}$-colors.

Definition 14. A $\mathfrak{D}$-*color assignment* over places P is a function $\texttt{color} : P \to \Sigma$ mapping each place $p \in P$ to a corresponding $\mathfrak{D}$-color. □

As for control places, it is well-known that the coloring mechanism can be exploited to realize a plethora of conceptual abstractions on top of the control flow. We mention here the two most important abstractions in our setting: *(i)* cases and their data, and *(ii)* resource. A *case* represents a specific process instance, and its *case data* [33] are local data whose scope is the case itself, and that are used to store important information for the progression of the case. Such data may be either extracted from the underlying persistence layer, or obtained by interacting with the external environment (e.g., human users, external services, or data generators). *Resources* represent actors able to handle the execution of tasks. They are also typically associated to data attributes (e.g., id, role, group). Tasks typically consume (certain kinds of) resources when

[3] Hence, given a multiset S, we have $0 \cdot S = \emptyset$.

executed, and this implicitly affect the degree of concurrency in the progression of cases, as well as the possibility of spawning new cases.

The fact that control places are colored implies that whenever a token is assigned to a control place, it must carry a data tuple whose types match component-wise the place color. It is worth noting that a colored place may be interchangeably considered as a specific state/condition within the control layer, or as a special relation schema used to enrich the persistence layer with control-related information. Similarly, a token distributed over a place may be interchangeably seen as a thread of control located in that state, or as a tuple assigned to the relation schema represented by that place.

As discussed above, control places host tokens carrying local data. Obviously, the control layer also requires to query persistent data, using them to decide how to route tokens when it comes to business decisions, or to assign them to case data. We want to support both possibilities, but clearly separating the data retrieved from the persistence layer, from those carried by tokens. This is why we distinguish view places from control places. Each view place exposes to the control layer a portion of the data stored in the persistence layer. Formally, this is done by equipping the view place with a query defined in the data logic layer.

Definition 15. Given a data logic layer $\mathcal{L} = \langle \mathcal{Q}, \mathcal{A} \rangle$, a *query assignment* from view places P_v to queries $\mathcal{Q}$ is a function $\texttt{query} : P_v \to \mathcal{Q}$ mapping each view place $p \in P_v$ with $\texttt{color}(p) = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ to a query $Q(x_1, \ldots, x_n)$ from $\mathcal{Q}$, such that the color of p component-wise matches with the types of the free variables in Q: for each $i \in \{1, \ldots, n\}$, we have $\mathcal{D}_i = \texttt{type}(x_i)$. □

A view place may be seen as a normal place, whose color is implicitly obtained by the types of the free variables of the query, considered with their natural ordering. However, tokens are not arbitrarily attached to it: at a given time, the tokens it contains represent the answers to the query it is associated to. All such tokens are only "virtually" present in the control layer, and in fact they cannot be consumed within the control layer itself, but only accessed in a read-only way. Notice, however, that the content of the view place is not immutable: it changes whenever the data it fetches from the persistence layer are updated.

Example 9. Consider the db-net of Fig. 2. Place *FreeDrivers* is a view place, connected to the query $\texttt{Q}_{\texttt{FreeTaxi}}$ shown in Example 3. At a given time, such a place "inspects" the content of the underlying persistence layer and retrieves all pairs $\langle tid, pn \rangle$, where *tid* is the id of a free taxi, and *pn* is its plate number. Such pairs are seen as tokens "virtually" present in the view place. Place *ReservedTaxi* is a normal, control place, used to store session ids together with their corresponding reserved taxi id. ■

Transitions. As customary, in our model transitions represent atomic units of work within the control layer, thus providing the fundamental building block to describe the dynamics of a process. In our setting, they simultaneously account for three different aspects: the token consumption/production mechanism of

CPNs, the injection of possibly fresh data from the external environment a là ν-Petri nets [32], and the impact on the underlying persistence layer.

We start with the consumption of tokens. This is modeled through input arcs connecting places to transitions, together with inscriptions that declaratively match tokens and their data. To this end, we build on the approach adopted in variants of data nets [5,25,32]: an inscription is just a multiset of tuples over a given set of typed variables. Each tuple nondeterministically matches a token from the input place, and the variables therein are bound, component-wise, to the data carried by that token. Upon firing, the token is consumed if the input place is a control place, whereas it is *only inspected* if the place is a view place.

The overall consumption/inspection of tokens and the data they carry along all arcs incoming into a transition constitutes a *firing mode* for that transition. In the context of a transition definition, we call *inscription* a tuple of typed variables (and, possibly, values). We denote the set of all possible inscriptions over set $\mathcal{Y}$ as $\Omega_{\mathcal{Y}}$, and the set of variables appearing inside an inscription $\omega \in \Omega_{\mathcal{Y}}$ as $Vars(\omega)$, extending such notation to sets and multisets of inscriptions.

Definition 16. An *input flow* from places P to transitions T is a function $F_{in} : P \times T \to \Omega^{\oplus}_{\mathcal{V}_{\mathfrak{D}}}$ assigning multisets of inscriptions (over variables $\mathcal{V}_{\mathfrak{D}}$) to input arcs, such that all such inscriptions are compatible with their input places. An inscription $\langle x_1, \ldots, x_m \rangle$ is *compatible* with a place p if $\mathtt{color}(p) = \mathcal{D}_1 \times \ldots \times \mathcal{D}_m$, such that for every $i \in \{1, \ldots, m\}$, we have $\mathtt{type}(x_i) = \mathcal{D}_i$. □

Graphically, we do not depict input arcs whose inscription is $\emptyset$. We define the *input variables* of t, written $InVars(t)$ as the set of all variables occurring on input arc inscriptions for t:

$$InVars(t) = \{x \in \mathcal{V}_{\mathfrak{D}} \mid \text{there exists } p \in P \text{ such that } x \in Vars(F_{in}(\langle p, t \rangle))\}.$$

The set $InVars(t)$ gives an indication about which input data elements are accessed when a transition fires. The multiple usage of the same variable in an inscription, or in inscriptions attached to different arcs incident to a transition, captures the requirement of *matching* the same data object in different tokens, allowing the transition to fire only if the accessed tokens carry the *same* data value. This mirrors the notion of join used when querying relational data. In general, though, the modeler may require to specify additional constraints over such input data to allow firing the transition. To this end, we introduce guards.

Definition 17. A $\mathfrak{D}$-typed *guard* is a formula of the form:

$$\varphi \ ::= \ \mathsf{true} \mid S(\vec{y}) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$$

where, for $\vec{y} = \langle y_1, \ldots, y_n \rangle \subseteq \mathcal{V}_{\mathfrak{D}}$, we have that S/n is a predicate defined in $\Gamma_{\mathcal{D}}$ for some $\mathcal{D} \in \mathfrak{D}$, and for each $i \in \{1, \ldots, n\}$, we have that y_i is either a value $\mathtt{o} \in \Delta_{\mathcal{D}}$, or a variable $x_i \in \mathcal{V}_{\mathfrak{D}}$ with $\mathtt{type}(x_i) = \mathcal{D}$. □

We denote by $\mathbb{F}_{\mathfrak{D}}$ the set of all possible $\mathfrak{D}$-typed guards. Additionally, with a slight abuse of notation, given guard φ we denote by $Vars(\varphi)$ the set of variables occurring in φ. Guards may be seen as the quantifier- and relation-free

fragment of $\texttt{FO}(\mathfrak{D})$ queries (cf. Definition 4). Consequently, their semantics is inherited from Definition 7 (considering the empty database instance). Guards are attached to transitions, and defined over their input variables, thus being an additional filter on the data that can be matched to the input inscriptions.

Definition 18. A $\mathfrak{D}$-typed *transition guard assignment* over transitions T is a function $\texttt{guard} : T \to \mathbb{F}_{\mathfrak{D}}$ assigning to each transition $t \in T$ a $\mathfrak{D}$-typed guard φ, such that $Vars(\varphi) \subseteq InVars(t)$. □

We now concentrate on the effect of firing a transition, which may simultaneously impact the control layer and the underlying persistence layer. Such an effect is tuned by the input variables attached to the transition, as well as additional data obtained from the external environment. Injection of external data is crucial for two reasons [5,11,28]. First, during the execution of a case, input data may be dynamically acquired from human users or external services, and used later on; this is, e.g., what happens when a user form needs to be filled before continuing with the case execution, then deciding how to route the case depending on the inserted data. Second, fresh ids may be injected into the system, e.g., to explicitly distinguish tokens via certain data attributes, or to insert a new tuple in the underlying database instance (which typically requires to create a distinctive primary key for that tuple). We call these two types of external inputs *arbitrary external inputs* and *fresh external inputs*. To account for arbitrary external inputs in the context of a transition, we just employ "normal" variables distinct from those used in the input inscriptions. To account for fresh external inputs, we employ the well-known mechanism adopted in ν-Petri nets [29,32]. In particular, we introduce a countably infinite set $\Upsilon_{\mathfrak{D}}$ of $\mathfrak{D}$-typed *fresh variables*. To guarantee an unlimited provisioning of fresh values, we impose that for every variable $\nu \in \Upsilon_{\mathfrak{D}}$, we have that $\Delta_{\texttt{type}(\nu)}$ is countably infinite.

From now on, we fix a countably infinite set of $\mathfrak{D}$-typed variable $\mathcal{X}_{\mathfrak{D}}$, obtained as the disjoint union of "normal" variables $\mathcal{V}_{\mathfrak{D}}$ and fresh variables $\Upsilon_{\mathfrak{D}}$. In formulae, $\mathcal{X}_{\mathfrak{D}} = \mathcal{V}_{\mathfrak{D}} \uplus \Upsilon_{\mathfrak{D}}$. Let us first focus on the impact of transition firing on the underlying persistence layer. This is, again, mediated by the data logic, exploiting in particular the actions it exposes. Specifically, a transition can bind to an action, using variables from $\mathcal{X}_{\mathfrak{D}}$ as "actual" parameters. In this light, data passing from the control to the persistence layer is captured by re-using the same variable inside an input inscription and an action binding for the same transition. When the transition fires, actual parameters are substituted with concrete data values, instanating the action and allowing for its further invocation.

Definition 19. Given a data logic layer $\mathcal{L} = \langle \mathcal{Q}, \mathcal{A} \rangle$, an *action assignment* from transitions T to actions $\mathcal{A}$ is a partial function $\texttt{act} : T \to \mathcal{A} \times \Omega_{\mathcal{X}_{\mathfrak{D}} \cup \Delta_{\mathfrak{D}}}$, where $\texttt{act}(t)$ maps t to an action $\alpha \in \mathcal{A}$ together with a (binding) inscription compatible with α. An inscription $\langle y_1, \ldots, y_m \rangle$ is compatible with α if $\alpha{\cdot}\texttt{params} = \langle z_1, \ldots, z_m \rangle$ and, for each $i \in \{1, \ldots, m\}$, we have $\texttt{type}(y_i) = \texttt{type}(z_i)$ if y_i is a variable from $\mathcal{X}_{\mathfrak{D}}$, or $y_i \in \Delta_{\texttt{type}(z_i)}$ if y_i is a value from $\Delta_{\mathfrak{D}}$. □

The action assignment provides a distinctive feature of our model, namely the ability of the control layer to invoke an action applied to the underlying persis-

tence layer. This, however, does not in general guarantee that the action invocation will actually turn into an update over the persistence layer. Recall in fact that an action instance is applied transactionally: if it produces a new database instance that is compliant with the persistence layer, the action instance *succeeds* and the update is committed; if, instead, some constraints is violated, the action instance *fails* and the update does not take place.

Lastly, we consider the effect of transitions on the control layer itself, defining which tokens have to produced, together with the data they will carry, and to which places such tokens have to be assigned. This is done by mirroring the definition of input flow (cf. Definition 16), with two distinctions. First, output arcs connect transitions to control places only, as view places cannot be explicitly modified within the control layer. Second, the inscriptions attached to output arcs may mention not only input variables, but also: *(i)* values, allowing for constructing tokens that carry explicitly specified data; *(ii)* fresh variables, allowing for constructing tokens that carry data not already present in the net, nor in the underlying database instance.

Definition 20. An *output flow* from transitions T to control places P_c is a function $F_{out} : T \times P_c \to \Omega^{\oplus}_{\mathcal{X}_\mathfrak{D} \cup \Delta_\mathfrak{D}}$ assigning multisets of inscriptions to output arcs, such that all such inscriptions are compatible with their output places (as defined in Definition 16). □

We do not depict output arcs graphically when their inscription is $\emptyset$. We define the *output variables* of t, written $OutVars(t)$, as the set of variables occurring in the action assignment for t (if any), and in its output arc inscriptions:

$$\begin{aligned} OutVars(t) = \quad & \{x \in \mathcal{X}_\mathfrak{D} \mid \mathtt{act}(t) \text{ is defined as } \langle \alpha, \omega \rangle, \text{ and } x \in Vars(\omega)\} \\ & \cup \{x \in \mathcal{X}_\mathfrak{D} \mid \text{there exists } p \in P \text{ such that } x \in Vars(F_{out}(\langle t, p \rangle))\}. \end{aligned}$$

With this notion at hand, we can obtain the *external variables* of transition t as $OutVars(t) \setminus InVars(t)$. Each such variable x is not bound by any input inscription, and can consequently be assigned arbitrarily (if $x \in \mathcal{V}_\mathfrak{D}$), or to a fresh value (if $x \in \Upsilon_\mathfrak{D}$). Among such variables, we explicitly refer to the fresh variables attached to t, using notation $FreshVars(t)$. Mathematically, $FreshVars(t) = OutVars(t) \cap \Upsilon_\mathfrak{D}$.

Example 10. Consider the *FinalizeBooking* transition in Fig. 2. It has three input arcs, used to consume three tokens respectively belonging to three places *ReservedTaxi*, *PhoneNumber*, and *PickupData*. The inscriptions on the input arcs indicate that whenever three tokens from such places are consumed, they have to agree on their first data component, i.e., they must belong to the same session. This realizes a sort of join, and ensures that only tokens produced within the same session are considered upon firing. The so-obtained data are then fed to the AddBooking action, which uses the session id *sid*, the reserved taxi id *tid*, the phone number n, the address a and the time t to create a new booking. However, since the creation of a new booking also requires to provide a fresh id for a new tuple to be inserted in the *Pickup_Data* relation schema, an additional fresh variable ν_{pdid} is also used when invoking that action. ■

As discussed before, firing a transition may incur in the instantiation and invocation of an action from the data logic layer, and the so-obtained action instance may or not result in an actual update. To raise awareness of the control layer about these two radically different outcomes, we introduce two separate output flows: a normal output flow, capturing the actual effect of a transition on the control flow when its attached action succeeds, and a *rollback flow*, capturing the actual effect of a transition on the control flow when its attached action fails. With this distinction, the control layer can fine-tune its own behavior in accordance with the transactional semantics of the persistence layer, e.g., taking a standard or a compensation route depending on the outcome of the action. To graphically distinguish *normal output arcs* from *rollback output arcs*, we proceed as follows. We depict the former as usual:  . Instead, we decorate the latter with an "x": .

Definition 21. A $\mathfrak{D}$-typed *control layer* over a data logic layer $\mathcal{L} = \langle \mathcal{Q}, \mathcal{A} \rangle$ is a tuple $\langle P, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \texttt{guard}, \texttt{act} \rangle$, where:

- $P = P_c \uplus P_v$ is a finite set of control places constituted by control places P_c and view places P_v;
- T is a finite set of transitions, such that $T \cap P = \emptyset$;
- F_{in} is an input flow from P to T (cf. Definition 16);
- F_{out} and F_{rb} are two output flows from T to P_c (cf. Definition 20), respectively called *normal output flow*and *rollback flow*;
- `color` is a color assignment over P (cf. Definition 14);
- `query` is a query assignment from P_v to $\mathcal{Q}$ (cf. Definition 15);
- `guard` is a transition guard assignment over T (cf. Definition 18);
- `act` is an action assignment from T to $\mathcal{A}$ (cf. Definition 19). □

3.4 DB-nets

We now put the three layers together, providing a formal definition for db-nets.

Definition 22. A *db-net* is a tuple $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N} \rangle$, where:

- $\mathfrak{D}$ is a type domain (cf. Definition 1);
- $\mathcal{P}$ is a $\mathfrak{D}$-typed persistence layer (cf. Definition 9);
- $\mathcal{L}$ is a $\mathfrak{D}$-typed data logic layer over $\mathcal{P}$ (cf. Definition 13);
- $\mathcal{N}$ is a $\mathfrak{D}$-typed control layer over $\mathcal{L}$ (cf. Definition 21). □

Example 11. Figure 3 shows the control layer of a db-net $\mathcal{B}$, using the persistence layer $\mathcal{P}$ defined in Example 5 and the data logic layer $\mathcal{L}$ defined in Example 8. The control layer realizes a simple ticket processing workflow, where tickets are created, manipulated, and finally resolved. In spite of its simplicity, $\mathcal{B}$ already shows many distinctive features of our model. We intuitively describe the control layer moving from left to right and from top to bottom. Each case of this process is constituted by a ticket and its responsible employee. A ticket is created by the Create Ticket transition, which requires the presence of an idle

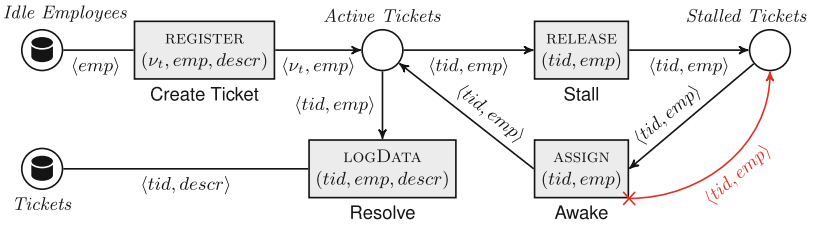


Fig. 3. The control layer of a db-net for ticket management. In CreateTicket, ν_t is a fresh input variable, and *descr* is an arbitrary input variable.

employee to be fired. Since this condition needs to inspect the persistence layer so as to retrieve idle employees, we model it through a view place associated to query $\mathtt{Q_e}$ from $\mathcal{L}$. Notice that if no employee is currently idle, then Create Ticket is not enabled. Upon firing Create Ticket for a given idle employee, a fresh ticket id is generated using fresh variable ν_t, and a ticket description is obtained through the "external" input variable *descr*. All such data are bound to action REGISTER, which is applied when the transition fires. Among the effects of REGISTER, there is one asserting that the selected employee becomes responsible for the newly created ticket. This indirectly implies that such an employee is not present anymore in the view place for idle employees. The ticket id, together with its responsible employee, represent the case and its data. The two control places *Active Tickets* and *Stalled Tickets* have color **int** $\times$ **string**, and model two distinct states in which tickets may be. Such states are important only within the evolution of cases, and are therefore not propagated to the underlying persistence layer. An active ticket may be "stalled" if the employee is currently unable to resolve it. Executing the Stall transition has a twofold effect. Within the control layer, the ticket is moved from active to stalled. Within the persistence layer, its responsible employee is released. Interestingly, the relation of responsibility is now only recalled within the control layer. A stalled ticket may be revived, by inserting such a relation back into the persistence layer. This is captured by the Awake transition, which mirrors the effect of the Stall transition. However, there is a particularly interesting aspect here. When a ticket t_1 is stalled, its responsible employee e is released and becomes idle. She may be then selected as responsible of a newly created ticket t_2. Due to the constraints present in $\mathcal{P}$, the indirect effect of this situation is that t_1 cannot be awaken unless t_2 is either stalled or resolved. In fact, awakening t_1 in a situation where t_2 is active would violate the requirement that e is responsible of at most one ticket. For this reason, we enrich the Awake transition with a rollback output arc, which brings back the ticket to the stalled state if it is awaken in the "wrong" moment. For example, if t_1 is awaken while t_2 is active, the application of ASSIGN applied to $\langle t_1, e\rangle$ will fail, consequently bringing t_1 back to stalled. Finally, an active ticket may be resolved. This has a twofold effect. On the one hand, the token carrying the ticket and its responsible employee is removed from the net. On the other hand, the case information is logged into the persistence layer. However, logging also

requires to retrieve the description of the ticket. To this end, we employ a second view place accessing tickets and their description by exploiting $\mathtt{Q_t}$ from $\mathcal{L}$. By using the same variable *tid* in the two input inscriptions of the Resolve transition, we realize a join, thus inspecting the view place and extracting the description of *tid*.

4 Execution Semantics

The execution semantics of a db-net simultaneously accounts for the progression of a database instance compliant with the persistence layer of the net, and for the evolution of a marking over the control layer of the net. Such two information sources affect each other via the data logic layer: the database instance exposes its own data through view places, influencing the current marking and the enabled transitions; the marking over the control layer determines which transitions may be fired, in turn triggering updates the database instance. As customary in process analysis, the execution semantics considers the db-net of interest in *isolation*, hence assuming that the persistence layer of the db-net is updated only through its associated control layer, without any form of interference from external, unpredictable updates. Notice that external updates can be simulated in the db-net by means of additional, always enabled transitions, which may nondeterministically fire and modify the content of the persistence layer.

We start by formalizing the notion of marking over the control layer of the db-net. A marking distributes tokens over the places of the net, so that each token carries data that are compatible with the color of the place in which that token resides. In this light, tokens are nothing else than tuples of values over the place colors. In addition, the marking of a view place must correspond to the answers obtained by issuing its associated query over the underlying database instance.

Definition 23. A *marking* of a $\mathfrak{D}$-typed control layer $\mathcal{N} = \langle P, T, F_{in}, F_{out}, F_{rb}, \mathtt{color}, \mathtt{query}, \mathtt{guard}, \mathtt{act} \rangle$ is a function $m : P \to \Omega^{\oplus}_{\mathfrak{D}}$ mapping each place $p \in P$ to a corresponding multiset of p-compatible tuples using data values from $\mathfrak{D}$. A tuple $\langle \mathtt{o_1}, \ldots, \mathtt{o_n} \rangle$ is p-compatible if $\mathtt{color}(p)$ is of the form $\langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle$, and for every $i \in \{1, \ldots, n\}$, we have $\mathtt{o_i} \in \Delta_{\mathcal{D}_i}$. Given a database instance $\mathcal{I}$, we say that m is *aligned* to $\mathcal{I}$ via $\mathtt{query}$ if the tuples it assigns to view places exactly correspond to the answers of their corresponding queries over $\mathcal{I}$: for every view place $v \in P$ and every v-compatible tuple $\vec{\mathtt{o}}$, we have that $\vec{\mathtt{o}} \in m(v)$ if and only if $\vec{\mathtt{o}} \in \mathit{ans}(\mathtt{query}(v), \mathcal{I})$. □

We mirror the notion of active domain as provided in Definition 3 to the case of markings. Given a type $\mathcal{D} \in \mathfrak{D}$, the $\mathcal{D}$-*active domain* of a marking m, written $\mathit{Adom}_{\mathcal{D}}(m)$, is the set of values in $\Delta_{\mathcal{D}}$ such that $\mathtt{o} \in \mathit{Adom}_{\mathcal{D}}(m)$ if and only if there exists p such that $\mathtt{o}$ occurs in $m(p)$. From the practical point of view, one may consider the marking of control places to be initially defined by

the modeler, and then evolved by the control layer, while the marking of view places computed on-the-fly from the underlying database instance when needed.

In db-nets, then, both the persistence layer and the control layer are stateful: during the execution, the persistence layer is associated to a database instance, while the control layer to a marking aligned with that database instance.

Definition 24. A *snapshot* of a db-net $\mathcal{B} = \langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N} \rangle$ (also called $\mathcal{B}$-*snapshot*) is a pair $\langle \mathcal{I}, m \rangle$, where $\mathcal{I}$ is a database instance compliant with $\mathcal{P}$, and m is a marking of $\mathcal{N}$ aligned to $\mathcal{I}$ via $\mathtt{query}$. □

As customary for CPNs, the firing of a transition t in a snapshot is defined w.r.t. a so-called *binding* for t, that is, a substitution $\sigma : \mathit{Vars}(t) \to \Delta_{\mathfrak{D}}$, where $\mathit{Vars}(t) = \mathit{InVars}(t) \cup \mathit{OutVars}(t)$. However, to properly enable the firing of t, the binding σ must guarantee a number of properties:

1. agreement with the distribution of tokens over the places, in accordance with the inscriptions on the corresponding input arcs;
2. satisfaction of the guard attached to t;
3. proper treatment of fresh variables, guaranteeing that they are substituted with values that are pairwise distinct, and also distinct from all the values present in the current marking, as well as in the current database instance.

To formalize these conditions, we need a notion of *inscription binding*. Given an inscription (i.e., multiset of tuples of variables) $\omega \in \Omega^{\oplus}_{\mathcal{X}_{\mathfrak{D}} \cup \Delta_{\mathfrak{D}}}$, and a substitution θ over a set X of variables containing all variables from ω, the *inscription binding* of ω under θ is a multiset $\theta^{\oplus}(\omega)$ from $\Omega^{\oplus}_{\mathfrak{D}}$ defined as follows: $\langle \mathtt{o_1}, \ldots, \mathtt{o_n} \rangle^m \in \theta^{\oplus}(\omega)$ if and only if $\langle y_1, \ldots, y_n \rangle^m \in \omega$, such that for every $i \in \{1, \ldots, n\}$, we have $\mathtt{o_i} = y_i$ if $y_i \in \Delta_{\mathfrak{D}}$, or $\mathtt{o_i} = \theta(y_i)$ if $y_i \in \mathcal{X}_{\mathfrak{D}}$. For example, given $\omega = \{\langle x, y \rangle^2, \langle x, 1 \rangle\}$ and $\theta = \{x \mapsto 1, y \mapsto 2\}$, we have $\theta^{\oplus}(\omega) = \{\langle 1, 2 \rangle^2, \langle 1, 1 \rangle\}$.

Definition 25. Let $\mathcal{B}$ be a db-net with control layer $\langle P, T, F_{in}, F_{out}, F_{rb}, \mathtt{color}, \mathtt{query}, \mathtt{guard}, \mathtt{act} \rangle$. A transition $t \in T$ is *enabled* in a $\mathcal{B}$-snapshot $\langle \mathcal{I}, m \rangle$, written $\langle \mathcal{I}, m \rangle [t, \sigma \rangle$, if:

1. for every place $p \in P$, $m(p)$ provides enough tokens matching those required by inscription $\omega = F_{in}(\langle p, t \rangle)$ once ω is bound by σ, i.e., $\sigma^{\oplus}(\omega) \subseteq m(p)$;
2. $\mathtt{guard}(t)\sigma$ is true;
3. σ is injective over $\mathit{FreshVars}(t)$, thus guaranteeing that fresh variables are assigned to pairwise distinct values by σ, and for every fresh variable $\nu \in \mathit{FreshVars}(t)$, $\sigma(\nu) \notin (\mathit{Adom}_{\mathtt{type}(\nu)}(\mathcal{I}) \cup \mathit{Adom}_{\mathtt{type}(\nu)}(m))$. □

Definition 26. Let $\mathcal{N} = \langle P, T, F_{in}, F_{out}, F_{rb}, \mathtt{color}, \mathtt{query}, \mathtt{guard}, \mathtt{act} \rangle$ be a $\mathfrak{D}$-typed control layer, and let $t \in T$ be a transition of $\mathcal{N}$ such that $\mathtt{act}(t) = \langle \alpha, \omega \rangle$, with $\alpha{\cdot}\mathtt{params} = \langle x_1, \ldots, x_n \rangle$ and $\omega = \langle y_1, \ldots, y_n \rangle$. The *action instance induced* by transition $t \in T$ under binding σ, written $\mathtt{act}_{\sigma}(t)$, is the action instance $\alpha\sigma'$, where $\sigma' : \alpha{\cdot}\mathtt{params} \to \Delta_{\mathfrak{D}}$ is a substitution for the formal parameters of α, defined as: for every $i \in \{1, \ldots, n\}$, if $y_i \in \Delta_{\mathfrak{D}}$, then $\sigma'(x_i) = y_i$; if instead $y_i \in \mathcal{X}_{\mathfrak{D}}$, then $\sigma'(x_i) = \theta(y_i)$. □

The firing of an enabled transition under some mode has then a threefold effect. First, all tokens present in control places that are used to match the input inscriptions are consumed. Second, the action instance induced by the firing is applied on the current database instance. If the application is successful, the database instance is updated (*commit*); if not, it is kept unaltered (*rollback*). Third, tokens built from the inscriptions on output arcs are produced and put into target places, considering either normal output arcs or rollback arcs depending on whether the action instance has been committed or rolled back.

Definition 27. Let $\mathcal{B} = \langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N} \rangle$ be a db-net with $\mathcal{N} = \langle P, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \texttt{guard}, \texttt{act} \rangle$, and $s_1 = \langle \mathcal{I}_1, m_1 \rangle$, $s_2 = \langle \mathcal{I}_2, m_2 \rangle$ be two $\mathcal{B}$-snapshots. Let $t \in T$ be a transition of $\mathcal{N}$, and σ be a binding for t, such that $s_1[t, \sigma\rangle$. We say that t *fires* in s_1 with binding σ producing s_2, written $s_1[t, \sigma\rangle s_2$, if the following conditions hold: given $\mathcal{I}_3 = \texttt{apply}(\texttt{act}_\sigma(t), \mathcal{I}_1)$,

- if $\mathcal{I}_3$ is compliant with $\mathcal{P}$, then $\mathcal{I}_2 = \mathcal{I}_3$, otherwise $\mathcal{I}_2 = \mathcal{I}_1$;
- For every control place $p \in P$, given $\omega_{in} = F_{in}(\langle p, t \rangle)$, $\omega_{out} = F_{out}(\langle t, p \rangle)$, and $\omega_{rb} = F_{rb}(\langle t, p \rangle)$, we have

$$m_2(p) = (m_1(p) - \sigma^{\oplus}(\omega_{in})) + k_{out} \cdot \sigma^{\oplus}(\omega_{out}) + (1 - k_{out}) \cdot \sigma^{\oplus}(\omega_{rb}),$$

  where $k_{out} = 1$ if $\mathcal{I}_3$ is compliant with $\mathcal{P}$, and $k_{out} = 0$ otherwise. □

The execution semantics of a db-net is captured by a possibly *infinite-state labeled transition system* (LTS) that accounts for all possible executions of the control layer starting from an initial snapshot. States of this transition systems are db-net snapshots, and transitions model the effect of firing db-net transitions under given bindings. Formally, given a db-net $\mathcal{B} = \langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N} \rangle$ with $\mathcal{N} = \langle P, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \texttt{guard}, \texttt{act} \rangle$, and given a snapshot s_0 over $\mathcal{B}$ (called the *initial $\mathcal{B}$-snapshot*), the *execution semantics* of $\mathcal{B}$ starting from s_0 is given by the LTS $\Gamma^{\mathcal{B}}_{s_0} = \langle S, s_0, \rightarrow \rangle$, where:

- S is a possibly infinite set of $\mathcal{B}$-snapshots;
- $\rightarrow \subseteq S \times T \times S$ is a *transition relation* over states, labeled by transitions T;
- S and $\rightarrow$ are defined by simultaneous induction as the smallest sets such that: *(i)* $s_0 \in S$; *(ii)* given a $\mathcal{B}$-snapshot $s \in S$, for every transition $t \in T$, binding σ, and $\mathcal{B}$-snapshot s', if $s[t, \sigma\rangle s'$ then $s' \in S$ and $s \xrightarrow{t} s'$.

Example 12. Consider the db-net $\mathcal{B}$ in Fig. 3, with the initial $\mathcal{B}$-snapshot s_1 that contains two idle employees `Paul` and `Jane`. Figure 4 shows a possible, finite execution of $\mathcal{B}$ starting from s_1, which actually corresponds to a portion of the LTS $\Gamma^{\mathcal{B}}_{s_1}$. For example, to reach s_2 from s_1, Create Ticket has to be fired with binding $\sigma = \{\nu_t \mapsto \texttt{12}, emp \mapsto \texttt{Jane}, descr \mapsto \text{☎}\}$. The resulting snapshot contains the generated ticket, and is such that `Jane` is not idle anymore. Notably, along the run we encounter a situation that concretely demonstrates the rollback semantics. This is the case of snapshot s_4, where `Jane` is responsible of an active, but also associated to a currently stalled ticket. Due to the constraints present

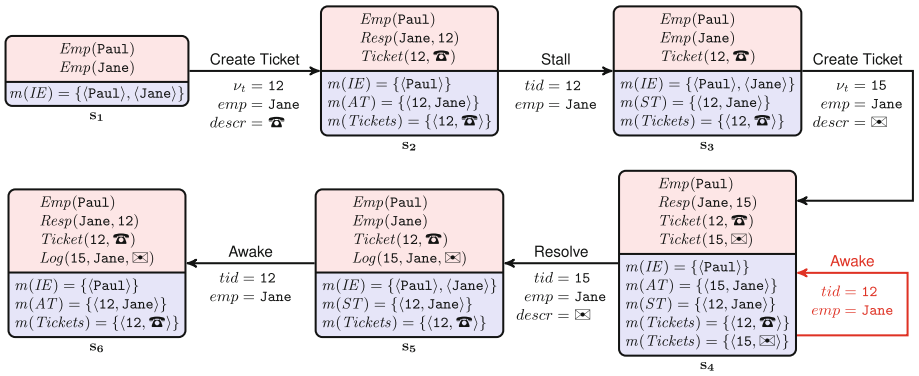


Fig. 4. A finite execution of the db-net from Example 11. Here we use the following abbreviations: $AT = Active\ Tickets$, $IE = Idle\ Employees$, $ST = Stalled\ Tickets$.

in the persistence layer of $\mathcal{B}$, this implies that `Jane` cannot awake the stalled ticket unless the active one is either stalled or resolved. Hence, if `Jane` chooses to awake the stalled ticket, the system ends up into an inconsistent state due to the violation of the database constraint imposting that an employee can be responsible of at most one ticket (cf. Example 5). This leads to the rollback of ASSIGN, and to the consequent activation of the corresponding rollback arc (marked in red in Fig. 4). In this specific example, the rollback arc has the effect of bringing the consumed ticket back to the stalled state, thus concretely realizing a simultaneous rollback for the persistence and the control layers.

5 Modeling and Verification

We discuss some key features of db-nets, considering the problem of modeling data-aware business processes, and that of verification, where we provide some initial results on the boundaries of decidability for the formal analysis of db-nets.

5.1 Modeling

From the modeling point of view, db-nets incorporate all typical abstractions needed in data-aware business processes, reconstructing all the distinctive features of various Petri net classes enriched with data, as well as those of data-centric processes. More formally, in terms of expressiveness, we observe the following correspondences. First of all, db-nets subsume ν-PNs [32], and become expressively equivalent to ν-PNs when: *(i)* there is only one unary color assigning places to the only one unordered countably infinite data type, *(ii)* the data logic layer is empty. With such a restrictions, the only modeling construct not natively provided by ν-PNs is that of arbitrary external input, which can be however simulated using ν-PNs by following the strategy defined in [5]. Second, db-nets are expressively equivalent to recently introduced formal models

for data-centric business processes, like DCDSs [9] and DMSs [5]. To transform those models into a db-net, it is sufficient to realize a control layer that simulates the application of condition-action rules. The translation of a db-net into those models, instead, is more convoluted, but can be attacked by leveraging the technique introduced in [29] to encode ν-PNs into DCDSs, with the only difference that while the transformation from ν-PNs into DCDSs requires the introduction of binary relations for places (keeping track of token ids and their pure names), in the case of db-nets the arity of place relations depend on the place colors.

Since DCDSs and DMSs are expressively equivalent to the richest models for business artifacts, such a correspondence paves the way towards the study of CPN-based business artifacts, making approaches like that of [27] data-aware. We also stress that db-nets go beyond the aforementioned approaches, since they conceptually componentize the different aspects of a dynamic system, giving first-class citizenships to relations, constraints, queries, database access points in the process, database updates triggered by the process, external inputs, and so on. This creates the basis for studying db-nets from the conceptual and methodological point of view, and exploit them to formalize concrete process-aware information systems like Bizagi, Bonita, and Camunda, in the style of [17].

5.2 Formal Verification

Formal verification of db-nets is obviously of utmost importance. As usual, verification may range from the analysis of fundamental properties such as *reachability*, to model checking temporal logics.

In the case of data-aware dynamic systems, such formal properties have also to incorporate a "first-order" component, consequently allowing one to inspect the data present in the system, and express properties about the (un)desired evolutions of such data [11,29]. We consider here a very pristine form of reachability, namely *reachability of a nonempty place*. Formally, this decision problem is defined as follows:

Input: A db-net $\mathcal{B}$, an initial $\mathcal{B}$-snapshot s_0, and a control place p from $\mathcal{B}$
Question: starting from s_0, is it possible to reach a $\mathcal{B}$-snapshot where p has at least one token? Formally, given the LTS $\Gamma^{\mathcal{B}}_{s_0} = \langle S, s_0, \rightarrow \rangle$, does there exist a finite sequence of $\mathcal{B}$-snapshots of the form $s_0 \rightarrow \ldots \rightarrow s_n = \langle \mathcal{I}_n, m_n \rangle$, such that $|m_n(p)| \geq 1$?

We show next that, as expected, db-nets are Turing-powerful even when the different layers are severely restricted, basic verification tasks such as reachability of a nonempty place remain undecidable.

Theorem 1. *Reachability of a nonempty place is undecidable:*

1. *for db-nets whose control layer only employs unary, string types, and whose data logic and persistence layers are empty;*
2. *for db-nets whose control layer consists of a state machine, and whose persistence layer contains only two unary relations.*

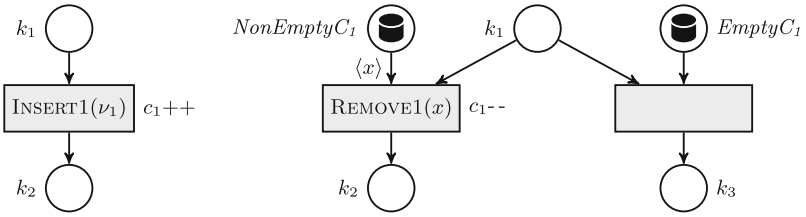
Proof (sketch). The first case corresponds to reachability in ν-PNs, which is undecidable [32]. The second case can be proved by reducing into the state reachability problem for deterministic two-counter automata. In fact, it is sufficient to show how to simulate an increment transition and a conditional decrement transition of a two-counter automaton. Figure 5 shows two fragments of a db-net whose control layer is a simple state machine, respectively simulating increment and conditional decrement transitions for a counter. We assume that the underlying persistence layer contains two initially empty unary relations C_1 and C_2 of type **string**. The size of such relations (i.e., the number of string values they contain) simulates the value of the two corresponding counters. In this light, incrementing the first counter amounts to inserting a fresh string into C_1; this is done by the action INSERT1. As for conditional decrement, we proceed as follows. We create two view places for the first counter. The first view place is called *NonEmptyC1* and used to retrieve the values contained in relation C_1. If there is at least one such value, it means that the counter is positive. The second view place, called *EmptyC1*, is a boolean view place associated to a query that tests whether relation C_1 does not contain any value. So, if there is a token in such a view place, it means that the first counter is zero. The conditional decrement transition is then easily realized by choosing which transition to take depending on the content of such view places. In the case of decrement, one value from C_1 is nondeterministically picked and passed as parameter to the REMOVE1 action, which removes it from C_1. This two fragments are simply replicated to handle increment and conditional decrement using relation C_2. □

To contrast this undecidability proof, we consider the case where the "size" of information maintained by the control layer and the persistence layer is suitably controlled.

Definition 28. Let $\mathcal{B}$ be a db-net, s_0 a $\mathcal{B}$-snapshot, and $\Gamma^{\mathcal{B}}_{s_0} = \langle S, s_0, \rightarrow \rangle$ the corresponding LTS. We say that $\mathcal{B}$ is *bounded w.r.t.* s_0 if

- $\mathcal{B}$ is *width-bounded*, i.e., there exists $b_1 \in \mathbb{N}$ such that for every $\mathcal{B}$-snapshot $\langle \mathcal{I}, m \rangle$ reachable from s_0 through $\rightarrow$, the number of distinct values assigned by m to the places of $\mathcal{B}$ is bounded by b_1:
- $\mathcal{B}$ is *depth-bounded*, i.e., there exists $b_2 \in \mathbb{N}$ such that for every $\mathcal{B}$-snapshot $\langle \mathcal{I}, m \rangle$ reachable from s_0 through $\rightarrow$, the number of tokens (possibly with the same values) assigned by m to the places of $\mathcal{B}$ is bounded by b_2.
- $\mathcal{B}$ is *state-bounded*, i.e., there exists $b_3 \in \mathbb{N}$ such that for every $\mathcal{B}$-snapshot $\langle \mathcal{I}, m \rangle$ reachable from s_0 through $\rightarrow$, we have $|\bigcup_{\mathcal{D} \in \mathfrak{D}} Adom_{\mathcal{D}}(\mathcal{I})| \leq b_3$. □

The first two conditions lift the two notions of boundedness introduced in [32] to the case of db-nets, while the third one is borrowed from the notion of state-boundedness in DCDSs [9]. Notice that a bounded $\mathcal{B}$ may still give raise to an infinite-state LTS, due to the insertion of fresh values into the boundedly many information slots available, and the fact that no restriction is imposed on the size of the type domains, from which external inputs are borrowed. In spite of this infinity, we are able to prove the following key result.



(a) Increment (b) Conditional decrement transition

Fig. 5. Simulation of an increment and conditional decrement transition of a two-counter machine using db-nets

Theorem 2. *Reachability of a nonempty place is decidable for bounded db-nets using* **string** *and* **real** *types.*

Proof (sketch). The proof is obtained by reconstructing, step-by-step, the translation technique proposed in [29], which encodes a ν-PN into a corresponding DCDS [9]. On the one hand, the translation has to be generalized to the case of tokens carrying tuples of data, and on the other hand, it has to merge the DCDS resulting from the translation, with the one natively provided by the data logic layer of the db-net of interest. In [29], it is shown that the translation does not only preserve the execution semantics of the original net, but also guarantees that if the input ν-PN is width- and depth-bounded, then the corresponding DCDS is state-bounded. This holds also in our setting, and consequently we inherit the decidability result for model checking first-order μ-calculus properties as considered in [29], which are clearly able to express reachability properties. Finally, the case of reals is handled as shown in [13], extending the abstraction technique of [9,29] so as to deal with dense orders. □

Interestingly, decidability holds not only for reachability, but also for the variant of first-order μ-calculus considered in [29], thus allowing one to check data-aware soundness [29] over db-nets. In addition, we conjecture that decidability can be strengthened to the case where the db-net of interest is width- and state-bounded, but not depth-bounded (the first two notions being essential towards decidability, as a consequence of Theorem 1).

Such initial undecidability and decidability results pave the way towards a refine analysis of the boundaries of decidability for the formal analysis of db-nets, taking advantage from the fact that the different sources of complexity are clearly separated in our model. For example, decidability could be studied by restricting the query language used in the data logic layer, or by leveraging recent dychotomic results on the analysis of data-aware extensions of Petri nets with ordered vs. unordered data types, and in presence or absence of (globally) fresh inputs, which are intimately connected to the boundaries of decidability for reachability [25]. In this light, it is important to notice that the technique mentioned in the proof sketch of Theorem 2 cannot be lifted to the case of integers, for which reachability turns out to be undecidable when the db-net is

bounded (as a consequence of the undecidability result for orders with successor shown in [13]). Another interesting line could be to leverage techniques based on under-approximations [5]. Since our first decidability result is intimately linked with boundedness of db-nets, it opens up another interesting line of investigation on how to check, or guarantee using modeling strategies, that a db-net is state-bounded, leveraging recent results [10,28,29,32].

Finally, db-nets pave the way towards the formal analysis of additional properties, which only become relevant when CPNs are combined with relational databases. We mention in particular two families of properties. The first is related to *rollbacks*, so as to check whether it is always (or never) the case that a transition induces a failing action. The second is related to the *true concurrency* present in a db-net, which may contain transitions that appear to be concurrent by considering the control layer in isolation, but have instead to be sequenced due to the interplay with the persistence layer (and its constraints).

6 Conclusion

We have introduced a formal model for data-aware business processes that, for the first time, combines an approach based on colored Petri nets with the standard relational model and transactional updates over it. We hope that db-nets will attract attention of researchers interested in the formal analysis of data-aware dynamic systems, as well as that of those interested in providing strong foundations for process-aware information systems and their concrete languages. We close by mentioning a further area of research that we consider of particular relevance for the database community. Since the control layer of db-nets is grounded on CPNs, all simulation techniques developed for CPNs can be seamlessly lifted to our setting. The result of a db-net simulation produces, as a by-product, a final database instance, populated through the execution of the control layer. The so-obtained database instance implicitly reflects the footprint of the control layer, which inserts data as the result of the execution of a process. This makes the obtained database instance much more intriguing than one synthetically generated without considering how data are produced over time. In this light, simulation of db-nets has the potential of providing novel insights into the problem of data benchmarking [24], especially in the context of data preparation for process mining [2,14].

Acknowledgements. This research has been partially supported by the Euregio IPN12 *"KAOS: Knowledge-Aware Operational Support"* project, which is funded by the "European Region Tyrol-South Tyrol-Trentino" (EGTC) under the first call for basic research projects, and by the UNIBZ internal project *"KENDO (Knowledge-driven ENterprise Distributed cOmputing)"*.

References

1. Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997). doi:10.1007/3-540-63139-9_48

2. Aalst, W.M.P.: Process cubes: slicing, dicing, rolling up and drilling down event data for process mining. In: Song, M., Wynn, M.T., Liu, J. (eds.) AP-BPM 2013. LNBIP, vol. 159, pp. 1–22. Springer, Cham (2013). doi:10.1007/978-3-319-02922-1_1
3. van der Aalst, W.M.P., Stahl, C.: Modeling Business Processes - A Petri Net-Oriented Approach. Cooperative Information Systems series. MIT Press, Cambridge (2011)
4. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data Knowl. Eng. **53**(2), 129–162 (2005)
5. Abdulla, P.A., Aiswarya, C., Atig, M.F., Montali, M., Rezine, O.: Recency-bounded verification of dynamic database-driven systems. In: Proceedings of PODS. ACM Press (2016)
6. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley, Redwood City (1995)
7. Abiteboul, S., Segoufin, L., Vianu, V.: Static analysis of active XML systems. ACM Trans. Database Syst. **34**(4), 23 (2009)
8. Badouel, E., Hélouët, L., Morvan, C.: Petri nets with semi-structured data. In: Proceedings of PN, LNCS. Springer (2015)
9. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proceedings of PODS, pp. 163–174. ACM (2013)
10. Bagheri Hariri, B., Calvanese, D., Deutsch, A., Montali, M.: State boundedness in data-aware dynamic systems. In: Proceedings of KR (2014)
11. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: a database theory perspective. In: Proceedings of PODS (2013)
12. Calvanese, D., Giacomo, G., Montali, M., Patrizi, F.: Verification and synthesis in description logic based dynamic systems. In: Faber, W., Lembo, D. (eds.) RR 2013. LNCS, vol. 7994, pp. 50–64. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39666-3_5
13. Calvanese, D., Delzanno, G., Montali, M.: Verification of relational multiagent systems with data types. In: Proceedings of AAAI (2015)
14. Calvanese, D., Montali, M., Syamsiyah, A., Aalst, W.M.P.: Ontology-driven extraction of event logs from relational databases. In: Reichert, M., Reijers, H.A. (eds.) BPM 2015. LNBIP, vol. 256, pp. 140–153. Springer, Cham (2016). doi:10.1007/978-3-319-42887-1_12
15. Cohn, D., Hull, R.: Business artifacts: a data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. **32**(3), 3–9 (2009)
16. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. Inf. Syst. **38**(4), 561–584 (2013)
17. De Masellis, R., Di Francescomarino, C., Ghidini, C., Montali, M., Tessaris, S.: Add data into business process verification: bridging the gap between theory and practice. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of AAAI, pp. 1091–1099 (2017)
18. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proceedings of ICDT, pp. 252–267 (2009)
19. Dumas, M.: On the convergence of data and process engineering. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 19–26. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23737-9_2

20. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: Dfl: a dataflow language based on petri nets and nested relational calculus. Inf. Syst. **33**(3), 261–284 (2008)
21. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: Proceedings of ODBASE, pp. 1152–1163 (2008)
22. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009). doi:10.1007/b95112
23. Künzle, V., Weber, B., Reichert, M.: Object-aware business processes: fundamental requirements and their support in existing approaches. Int. J. Inf. Syst. Model. Des. **2**(2), 19–46 (2011)
24. Lanti, D., Rezk, M., Xiao, G., Calvanese, D.: The NPD benchmark: Reality check for OBDA systems. In: Proceedings of EDBT, pp. 617–628 (2015). OpenProceedings.org
25. Lasota, S.: Decidability border for petri nets with data: WQO dichotomy conjecture. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016. LNCS, vol. 9698, pp. 20–36. Springer, Cham (2016). doi:10.1007/978-3-319-39086-4_3
26. Libkin, L.: Elements of Finite Model Theory, LNCS, vol. 7360, chap. Fixed Point Logics and Complexity Classes. Springer (2004)
27. Lohmann, N.: Compliance by design for artifact-centric business processes. Inf. Syst. **38**(4), 606–618 (2013)
28. Montali, M., Calvanese, D.: Soundness of data-aware, case-centric processes. Int. J. Software Tools Technol. Transf. **18**, 535–558 (2016)
29. Montali, M., Rivkin, A.: Model checking petri nets with names using data-centric dynamic systems. Formal Aspects Comput. **28**(4), 615–641 (2016)
30. Reichert, M.: Process and data: two sides of the same coin? In: Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I.F. (eds.) OTM 2012. LNCS, vol. 7565, pp. 2–19. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33606-5_2
31. Richardson, C.: Warning: don't assume your business processes use master data. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 11–12. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15618-2_3
32. Rosa-Velardo, F., de Frutos-Escrig, D.: Decidability and complexity of petri nets with unordered data. Theor. Comput. Sci. **412**(34), 4439–4451 (2011)
33. Russell, N., Hofstede, A.H.M., Edmond, D., der Aalst, W.M.P.: Workflow data patterns: identification, representation and tool support. In: Delcambre, L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, O. (eds.) ER 2005. LNCS, vol. 3716, pp. 353–368. Springer, Heidelberg (2005). doi:10.1007/11568322_23
34. Triebel, M., Sürmeli, J.: Homogeneous equations of algebraic petri nets. In: Proceedings of CONCUR, pp. 1–14. LNCS, Springer (2016)
35. Vianu, V.: Automatic verification of database-driven systems: a new frontier. In: Proceedings of ICDT, pp. 1–13 (2009)